

# ENUMERATING THE SANEBLIDZE-UMBLE DIAGONAL TERMS

MIKAEL VEJDEMO-JOHANSSON

**ABSTRACT.** The author presents a computer implementation, calculating the terms of the Saneblidze-Umbler diagonals on the permutahedron and the associahedron. The code is analyzed for correctness and presented in the paper, the source code of which simultaneously represents both the paper and the program.

## 1. INTRODUCTION

In [SU04], Samson Saneblidze and Ron Umbler gave a combinatorial description of a diagonal on the permutahedron  $P_n$ , together with the associated diagonal on the associahedron  $K_n$  induced by the projection  $P_n \rightarrow K_n$  due to Andy Tonks [Ton97]. This provides a tool for describing an  $A_\infty$ -structure on the tensor product of two  $A_\infty$ -(co)algebras.

There has been work done by Steve Weaver, in [Wea05], to produce a computer implementation of the resulting algorithm for listing the terms. However, this implementation has not been widely disseminated, and further offers no transparency as to how the code written corresponds to the mathematical description. Furthermore, Andy Tonks [Ton07] has a private implementation of the resulting algorithm.

Haskell is one of the stricter functional programming languages. It also has very strong semantics. This means that a Haskell programmer can reason about the written program in a way that would yield a stringent proof of correctness. In this paper, the author presents a Haskell implementation of the enumeration algorithm of the Saneblidze-Umbler diagonal, interleaved with a description of the diagonal itself, in such a manner that the resulting code is clear enough and analogous enough to the mathematical argumentation for the correctness of the code to be clear.

In section 2, the author presents the code and the diagonal in parallel, beginning with the diagonal on the permutahedron and finalizing with a discussion and implementation of the projection. In section 3, a practical discussion of the usage of this paper for actual computation is given as well as an explicit example session with one of the mainstream Haskell interpreters. Finally, in section 4, an overview of the used symbols and library functions is given for reference for the readers.

## 2. THE SANEBLIDZE-UMBLE DIAGONAL

In this section we shall, in parallel, describe the diagonal on the permutahedron and the resulting diagonal on the associahedron and give computer code in the

---

The author acknowledges travel support from DFG Sachbeihilfe grant GR 1585/4-1.

functional programming language Haskell to work with computer calculations of the diagonal.

Since we shall be building a working programming library with code performing the relevant calculations, we need to declare the library as such.

```
module SaneblidzeUmbleSigns where
import Data.List
import Data.Maybe
import qualified Data.Map as Map
import Data.Map (!)
```

We shall be working a lot with representations of faces of the permutahedra  $P_n$ . These faces are indexed by ordered partitions, as defined in the next section. More specifically, we shall be spending most of our time working with  $C_*(P_n) = C_*(P_n; k)$  – the cellular chain complex of the  $n$ th permutahedron with coefficients in some field  $k$ . The aim of the efforts is to provide a cellular chain homotopic to a diagonal of  $P_n \times P_n$ , which thus will be given in  $C_*(P_n \times P_n) = C_*(P_n) \otimes C_*(P_n)$ .

**2.1. The diagonal on the permutahedron.** The enumeration given by Saneblidze and Umble [SU04] is obtained by defining various matrices and operations on those. The closure of the defined operations is in bijective correspondence with the terms of a diagonal on the permutahedron. This correspondence works by translating the matrices into pairs of ordered partitions, indexing a pair of faces on the permutahedron.

In order to make the exposition here closer to the programming, I shall omit further mention of matrices, and instead work exclusively with the partitions.

**Definition 2.1.** *An ordered partition of the ordered set  $\pi$  is a decomposition of  $\pi$  into disjoint ordered subsets  $\pi_1, \dots, \pi_n$  covering  $\pi$ . Such a decomposition is denoted by  $\pi_1 | \pi_2 | \dots | \pi_n$ .*

*Given a permutation  $\sigma \in S_n$ , the rising partition of  $\sigma$  of  $\sigma$  is a sequence of integer sequences, each of which is a maximal monotonically increasing subsequence of the  $\sigma(1), \dots, \sigma(n)$  and the concatenation of all sequences form the sequence  $\sigma(1), \dots, \sigma(n)$ .*

*Analogously, we define the falling partition of  $\sigma$  of a permutation.*

Since the process of finding the rising partition and the falling partition of  $\sigma$  has a common abstraction, we shall here give the code for that abstraction, and then the two relevant specializations.

```
type Sequence = [Int]
type Partition = [Sequence]

monotonicSequence :: (a → a → Bool) → [a] → [[a]]
monotonicSequence _ [] = []
monotonicSequence _ [x] = [[x]]
monotonicSequence cmp (x : y : etc) =
  if    x ‘cmp’ y
  then (x : s) : ss
  else [x] : (s : ss)
where
  (s : ss) = monotonicSequence cmp (y : etc)
```

$rising :: Sequence \rightarrow Partition$   
 $rising = monotonicSequence (\leq)$   
 $falling :: Sequence \rightarrow Partition$   
 $falling = monotonicSequence (\geq)$

Now, a basis element of the tensor product  $C_*(P_n) \otimes C_*(P_n)$  has the form  $u \otimes v$  where  $u$  and  $v$  are both partitions of the kind discussed here. Saneblidze and Umble prefer to order the partitions chosen for the diagonal in a way such that any operations defined on one side of the tensor product is reversed when performing it on the other. However, for ease of programming, we shall adopt a convention for internal representation in which the second argument is stored in reverse. Nevertheless, our output functions take this difference into consideration and make certain that everything displayed for the user adheres to Saneblidze and Umble's convention.

A permutation corresponds to a pair of partitions by reading the first partition as the falling sequences of the permutation, and the second as the rising sequences. This corresponds to reading columns and rows in the matrix representation used by Saneblidze and Umble. The matrix constructed directly from a permutation is called a *step matrix*, and we shall use this term to refer to the corresponding partition pairs throughout. The code that follows below incorporates this construction method.

Over fields of characteristic different from 2, the terms of the diagonal each have a sign. The code here developed will calculate the sign along with the terms, and optionally strip the signs when displaying the calculated faces. By the laziness properties of Haskell, these calculations will only be performed as and when the signs are requested.

**type** *Face* = (*Partition*, *Partition*)  
**type** *SignFace* = (*Int*, *Partition*, *Partition*)

$stripSign :: SignFace \rightarrow Face$   
 $stripSign (s, p, q) = (p, q)$

$buildFace :: Sequence \rightarrow SignFace$   
 $buildFace p = signFace (map\ sort\ (falling\ p), reverse\ (rising\ p))$

For an example consider the face denoted by Saneblidze and Umble as 12|34|5  $\otimes$  2|14|35. This is a step matrix resulting from the permutation 21435. In the internal representation it would be represented as  $([[1, 2], [3, 4], [5]], [[3, 5], [1, 4], [2]])$ . Without the function call to *sort* in the first component, it would have been  $([[2, 1], [4, 3], [5]], [[3, 5], [1, 4], [2]])$  and without the *reverse* in the second component, it would have been  $([[1, 2], [3, 4], [5]], [[2], [3, 5], [1, 4]])$ .

Thus, the benefit from the *sort* call is pure readability for the user, whereas the benefit from the use of *reverse* is in ease of code reuse, as will be seen when we implement the functions to generate derived matrices.

The formula for the sign is complicated. It is helpful to view it as the product of three different signs. One of these is calculated using the sign of a permutation. Suppose  $\sigma$  is a permutation in  $S_{n+1}$  giving rise to the face indexed by  $\mu' \otimes \mu$  with  $\mu = \mu_r | \dots | \mu_1$ . Then the concatenation of the parts of  $\mu$  retrieves the list of images for the partition  $\sigma$ . We define  $psgn(\sigma)$  as the usual permutation sign, and implement this as a simple orbit finder. The code given here adjusts for the fact that lists in Haskell have 0-based indexing.

$orbit :: Int \rightarrow Sequence \rightarrow Sequence$

$orbit\ a\ pi = findOrbit\ a\ []$

**where**

$findOrbit\ a\ as =$

**if**  $a' \in as$

**then**  $(sort \circ nub)\ (a : as)$

**else**  $findOrbit\ a'\ (a : as)$

**where**

$a' = pi\ !!\ (a - 1)$

$pSign :: Sequence \rightarrow Int$

$pSign\ pi = signPi$

**where**

$getOrbits\ orbs\ [] = orbs$

$getOrbits\ orbs\ (p : ps) = getOrbits\ (o : orbs)\ (ps \setminus o)$

**where**

$o = orbit\ p\ pi$

$orbits = getOrbits\ []\ pi$

$orbitLengths = map\ length\ orbits$

$evenCycles = filter\ even\ orbitLengths$

$signPi = (-1) \uparrow (length\ evenCycles)$

Furthermore, we let  $\epsilon = \sum_{i=1}^{r-1} i \cdot |\mu_i|$  and define the right-most partition sign  $sgn_r(\mu) = (-1)^\epsilon psgn(\sigma)$ .

$signR :: Partition \rightarrow Int$

$signR\ q = (-1) \uparrow epsilon * (pSign\ pi)$

**where**

$pi = concat\ q$

$epsilon = sum\ summands$

$summands = map\ (\lambda i \rightarrow i * (qLengths\ !!\ (i - 1)))\ [1..((length\ q) - 1)]$

$qLengths = map\ length\ q$

Finally, we define the order-reversing permutation sign by

$$orsgn(\mu) = (-1)^{\frac{1}{2}(\sum |U_i|^2 - (n+1))}$$

$orSign :: Partition \rightarrow Int$

$orSign\ p = (-1) \uparrow exponent$

**where**

$exponent = exponent2\ 'div'\ 2$

$exponent2 = (sum\ lengthSquares) - ((length \circ concat)\ p)$

$lengthSquares = map\ ((\uparrow 2) \circ length)\ p$

Following Saneblidze-Umble [SU04], we now allocate the sign

$$csgn(p \otimes q) = (-1)^{\binom{|q|}{2}} sgn_r(p) orsgn(q)$$

to a face  $p \otimes q$  given by a step matrix. We notice that in order to reconcile the internal representation used in this program with the conventions in [SU04], we further need to reverse the order of the parts in  $q$  to get the right sign.

$signFace :: Face \rightarrow SignFace$

$signFace\ (p, q) = (qSign * rSign * sign1, p, q)$

**where**

```

qSign      = (-1) ↑ qExp
qExp       = (choose2 ∘ length) q
rSign      = orSign p
sign1      = signR (reverse q)
choose2 n = n * (n - 1) `div` 2

```

Once the calculations have been performed, the user is likely to wish for a readable display of the calculated faces. In order to do this, the following code gives several functions that display faces of the permutahedron in a way familiar to the reader.

```

showSignFace :: SignFace → String
showSignFace f@(s, -, -) =
  case s of
    1   → "+" ++ (showFace ∘ stripSign) f
  - 1   → "-" ++ (showFace ∘ stripSign) f
    0   → ""
    a   → (show a) ++ "." ++ (showFace ∘ stripSign) f
showFace :: Face → String
showFace = showFaceTemplate showPartition
showFaceShort :: Face → String
showFaceShort = showFaceTemplate showPartitionShort
showFaceTemplate :: (Partition → String) → Face → String
showFaceTemplate showP (u, v) = showP u ++ "x" ++ (showP ∘ reverse) v
showPartitionShort :: Partition → String
showPartitionShort = filter (≠ ' ', ',') ∘ showPartition
showPartition :: Partition → String
showPartition p = pString p
where
  pString      = concat ∘ intersperse "|" ∘ partsStrings
  partsStrings = map (concat ∘ intersperse ", " ∘ map show)

```

```

showMatrix :: Face → String
showMatrix (f1, f2) = unlines $ map (showLine f1) f2

```

**where**

```

showLine a b = concatMap (flip showPoint b) a
showPoint a b =
  if intersect a b ≠ []
  then show $ head $ intersect a b
  else "."

```

Using this code, we can gain string representations of the face  $+12|34|5 \otimes 2|14|35$  as follows.

<code>showSignFace</code>	<code>+1,2 3,4 5x2 1,4 3,5</code>	<code>showMatrix</code>	<code>.35</code>
<code>showFace</code>	<code>1,2 3,4 5x2 1,4 3,5</code>		<code>14.</code>
<code>showFaceShort</code>	<code>12 34 5x2 14 35</code>		<code>2..</code>

We shall need to enumerate all step matrices, and thus, we need a function to enumerate all permutations in  $S_n$ . The following code takes care of this.

```
permutations :: Int → [Sequence]
permutations n = permuteList [1..n]
```

where

```
permuteList [] = []
permuteList [a] = [[a]]
permuteList l = concatMap (\x → map (x:) (permuteList (l \ [x]))) l
```

From these matrices, we then generate the closure under two operations: first downshift and then rightshift. These are, with the representation I have chosen, the same operation on either side of the tensor product. So it is sufficient to define a single operation on a partition.

**Definition 2.2.** Let  $\pi = \pi_1|\pi_2|\dots|\pi_k$  be a partition. A proper subset  $M$  of some part  $\pi_j$  in  $\pi$  with  $j < k$  is admissible with respect to a partition  $\mu = \mu_1|\mu_2|\dots|\mu_r$  if  $\min M > \max \pi_{j+1}$  and, supposing that  $\min M$  occurs in  $\mu_k$ , then  $\pi_{j+1} \cap \bigcup_{t=k}^r \mu_t = \emptyset$ .

An  $M$ -shift of a partition  $\pi$  with respect to a partition  $\mu$ , with  $M$  an admissible subset of  $\pi_j$  with respect to  $\mu$ , is the operation that returns the new partition  $\pi_1|\dots|\pi_j \setminus M|\pi_{j+1} \cup M|\dots|\pi_k$ .

Note that we only need to define the shift operations on the first factor of a face, since we can always conjugate it with the twist operation that interchanges factors in a tensor product.

Admissibility is not enough for a particular move to be permitted. There is also a condition on the sequence of moves that led to the partition  $\pi$ : we require that the moves progress through the matrix, so that if we have performed a move on an admissible subset  $M \subset \pi_i$ , then all subsequent moves have to occur on subsets  $M' \subset \pi_j$  with  $j > i$ .

The conditions placed on permissibility apart from the admissibility condition ensure that no matrix occurs as a derived matrix from more than one step matrix. This makes it clear that the process of generating new derived matrices will stop at some point. Thus, among the moves *not* permitted we find

$$\begin{pmatrix} & 1 & 2 \\ 4 & 5 & \\ 3 & & \end{pmatrix} \Rightarrow \begin{pmatrix} & 1 & 2 \\ 4 & & 5 \\ 3 & & \end{pmatrix} \Rightarrow \begin{pmatrix} & 1 & 2 \\ & 4 & 5 \\ 3 & & \end{pmatrix}$$

Checking admissibility for a certain subset of  $\pi_1$  is just a matter of constructing programmatic checks for all conditions in the definition, and then ensuring that all these conditions hold.

Since Haskell evaluates as lazily as possible we can get around the fact that certain conditions can only be defined once earlier conditions are seen to hold, since the resulting code will break evaluation as soon as a non-holding condition has been found. Thus, the latter conditions are not evaluated unless the earlier have been found to hold, thus making the latter conditions well-defined.

```
isAdmissible :: SignFace → [Int] → Bool
```

```
isAdmissible f@(⌞, pi, mu) m = admitted
```

where

```
mIntersectPi = map (intersect m) pi
```

```

partsWithM      = findIndices (( $\equiv$  length m)  $\circ$  length) mIntersectPi
mInUniquePart = (1  $\equiv$  length partsWithM)
j              = head partsWithM
jLessK        = j < length pi
pi_j          = pi !! j
pi_j1         = pi !! (j + 1)
properSubset  = mInUniquePart  $\wedge$  jLessK  $\wedge$  (length m < length pi_j)
minLargerMax = (minimum m > maximum pi_j1)
k             = fromJust (findIndex (minimum m  $\in$ ) mu)
mus          = concat (drop k mu)
allzero       = null (intersect pi_j1 mus)
admitted      = properSubset  $\wedge$  minLargerMax  $\wedge$  allzero

```

At this point, moving an admissible subset is a matter of assembling a new list with the appropriate parts modified to remove the set from one part and adjoin it to the next, and adjusting the sign accordingly. In order to figure out the sign, we thus need to move our elements in increasing order, one by one, and catch the sign changes as we do.

Suppose, following Saneblidze and Umble [SU04], that we move the single element  $x \in \pi_i$  to  $\pi_{i+1}$ , and our face is currently represented as  $\pi \otimes \mu$ . We define the upper and lower cuts by  $(a, S] = \{s \in S \mid s > a\}$  and  $[S, a) = \{s \in S \mid s < a\}$  respectively. Then

$$\text{csgn}(R_x \pi \otimes \mu) = -\text{csgn}(\pi \otimes \mu) \cdot (-1)^{|(x, \pi_i] \cup [\pi_{i+1}, x)|}$$

Down moves are simply right moves conjugated with a transposition, or in other words, this operation conjugated with the twist map  $a \otimes b \mapsto b \otimes a$ . Thus, we need only handle this case, since the other case follows symmetrically.

The function here written handles non-admissible subsets gracefully.

```

moveSubset :: SignFace  $\rightarrow$  [Int]  $\rightarrow$  Maybe SignFace

```

```

moveSubset f @( $\_$ , p',  $\_$ ) m =

```

```

  if      isAdmissible f m
  then    Just (foldl' moveElement f (sort m))
  else    Nothing

```

```

  where

```

```

    moveElement :: SignFace  $\rightarrow$  [Int]  $\rightarrow$  SignFace

```

```

    moveElement (s, p, q) e = (s', p', q)

```

```

    where

```

```

      (Just i)  = findIndex (e  $\in$ ) p
      pi        = p !! i
      pi1       = p !! (i + 1)
      pmoved    = (take i p) ++ [pi \ \ [e], pi1 ++ [e]] ++ (drop (i + 2) p)
      lowercut  = filter (>e) pi
      uppercut  = filter (<e) pi1
      expmoved  = length (lowercut ++ uppercut)
      (s', p') =
        if      e > maximum pi1

```

```

      then  $(-s * (-1) \uparrow \text{expmoved}, \text{pmoved})$ 
    else  $(s, p)$ 

```

Now, to construct the Saneblidze-Umble diagonal as the closure of the step matrices under these shift operations, we begin by enumerating all admissible subsets of a single partition. We need only to check such sets for admissibility that are proper subsets of some part and consist of elements larger than the maximal element in the next part, since otherwise they would not fulfill even the first few conditions.

```

admissiblesInPin :: Int → SignFace → [[Int]]
admissiblesInPin i f@(⊖, pi, ⊖) = filter (⊖ ∘ null) admissibleSets

```

where

```

admissibleSets =
  if i + 2 > length pi
  then []
  else filter (isAdmissible f) candidates
    where
      candidates = filter (⊖ ∘ null) (subsets large)
      large       = filter (>m) (pi !! i)
      m           = maximum (pi !! (i + 1))

```

```

subsets :: [a] → [[a]]
subsets [] = [[]]
subsets (a : as) = map (a:) (subsets as) ++ subsets as

```

Now that we can enumerate all admissible subsets in a partition, it is a simple matter to take a face and form all derived faces by generating more and more admissible subsets and moving these. We handle the moves in the second partition, as mentioned before, by conjugating the shift operation by the twist map. Valid derived faces are such that are reached by first moving subsets right, and then moving them down.

```

twist :: SignFace → SignFace
twist (s, a, b) = (s, b, a)

derivedFaces :: SignFace → [SignFace]
derivedFaces f@(⊖, p, q) = derivedRightQ 0 [] [f]

```

where

```

lp = length p
lq = length q
derivedRightQ i r [] =
  if i ≥ lp - 2
  then derivedDownQ 0 [] r
  else derivedRightQ (i + 1) [] r
derivedRightQ i r (s : ss) = derivedRightQ i (s : r) (ss ++ rights)
    where
      rights = mapMaybe (moveSubset s) (admissiblesInPin i s)
      derivedDownQ i d [] =
        if i ≥ lq - 2
        then d

```



```

else derivedDownQ (i + 1) [] d
derivedDownQ i d (s : ss) = derivedDownQ i (s : d) (ss ++ downs)
where
  s'           = twist s
  downs        = map twist twistedDowns
  twistedDowns = (mapMaybe (moveSubset s') (admissiblesInPin i s'))

```

The actual diagonal we return is a linear combination of faces. In order to implement this, we use the notion of a *Data.Map*. This is an associative array, indexing integer values representing coefficients using the faces as indexing keys.

```

type LinearCombination vectors = Map.Map vectors Int
showLinearCombination :: LinearCombination Face → String
showLinearCombination lc = concatMap showSignFace (signFaceList lc)
addSignFaces :: [SignFace] → LinearCombination Face
addSignFaces [] = Map.empty
addSignFaces ((s, p, q) : as) = Map.insertWith (+) (p, q) s (addSignFaces as)
signFaceList :: LinearCombination Face → [SignFace]
signFaceList lc = map (λ(p, q), s) → (s, p, q) (Map.toList lc)
permutahedronDiagonal :: Int → LinearCombination Face
permutahedronDiagonal n = (addSignFaces ∘ nub) deriveds
where
  deriveds = concatMap derivedFaces primitiveFaces
  primitiveFaces = map buildFace (permutations n)

```

**2.2. A diagonal on the associahedron.** Given the diagonal on permutahedra constructed above, we construct a diagonal on the associahedra by applying the projection due to Andy Tonks [Ton97]. In practice, this projection eliminates, in practice, all faces that contain partitions that are not *derived consecutive* in the sense that  $[\min \pi_j, \max \pi_j] \subset \bigcup_{i \leq j} \pi_i$ . for all  $\pi_j \in \pi$ .

This test is easily written, and thus the projection reduces to extracting the summands that pass the test.

```

derivedConsecutive :: Partition → Bool
derivedConsecutive pi = checkPartition [] pi
where
  checkPartition n [] = True
  checkPartition n (pij : pi') =
    if ( intersect n' range ≡ range )
    then checkPartition n' pi'
    else False
  where
    n' = sort (n ++ pij)
    range = [minimum pij .. maximum pij]
associahedronDiagonal :: Int → LinearCombination Face
associahedronDiagonal n =
  Map.filterWithKey checkFace (permutahedronDiagonal n)
where

```

$n$	$ \Delta_{P_n}(e^n) $	Execution time	Total allocation	Peak allocation
1	1	<0.005s	45.813k	28.617k
2	2	<0.005s	53.438k	28.617k
3	8	<0.005s	95.648k	28.617k
4	50	<0.005s	402.500k	28.617k
5	432	0.02s	3.987M	56.695k
6	4 802	1.63s	45.687M	1.631M
7	65 536	399.97s	1.000G	22.198M
8	1 062 882	93 965.64s	39.205G	342.704M
9	20 000 000	>400h	N/A	>3.000G

TABLE 1. Performance and calculations

*checkFace* (*f1*, *f2*) *s* =  
*derivedConsecutive f1*  $\wedge$  *derivedConsecutive* (*reverse f2*)

### 3. INSTALLATION, USAGE AND EXAMPLE CALCULATIONS

The way the paper is written, the source code of the program is the source code of the paper. It can be downloaded with the paper source code from [arXiv:0707.4399](https://arxiv.org/abs/0707.4399)

In order to use this program, a Haskell interpreter or compiler would have to be installed. I have used the Glasgow Haskell Compiler (GHC) version 6.6.1 [GHC99] to develop this, but it should run on any platform supporting the Haskell98 standard [JHA<sup>+</sup>99].

A typical worksession might look something like Example 1

**3.1. Performance data.** We have tested the code and its performance by calculating, subsequently, the diagonals on  $P_1, \dots, P_7$ . The results of our calculations as well as some complexity measurements can be found in Table 1. The tests were performed on a double Dual Core AMD Opteron(tm) Processor 270 with 16G RAM running OpenSuSE 10.2 with standard linux kernel version 2.6.18.

Given the garbage collection that GHC uses, there is a difference to be observed between the total amount of memory ever allocated, and the maximal amount of memory allocated at a single point in time. The measurements will state both.

### 4. HASKELL NOTATION

The paper uses Haskell code for the algorithm discussion, and for the benefit of the reader, we shall here discuss the notation we used to provide a dictionary and help elucidate the code.

**4.1. Types and definitions.** Haskell is a statically typed language, which means that any object has a type, and can only be used in an expression if the type matches all functions involved. Types can be constructed from a family of primitive types, of which we here see *Int* and *Bool* occurring.

*Int* is a machine word type, admitting integers up to about  $2^{32}$  before overflowing. Since the code runs into performance problems before the integers we use reach 10, this is not viewed as a problem at this point.

*Bool* is a truth value type, with the two valid elements *True* and *False*.

Using these types, we are then able to construct more complex types. The most important derived type templates we use are the list types, the *Maybe* type, maps

and the pair type. We shall deal with each of these constructions in a section of their own.

All functions we declare have their type signature given explicitly. This is done by a declaration on the form

*functionName* :: *firstArgType* → *secondArgType* → ... → *returnType*

The function body then is declared by giving names to the function arguments, and declaring what happens with them. While doing this, there are two specific constructions with special meaning. First, there is the `_` variable name, which means that that particular input variable is not named in the coming declaration. This is, for instance used, for cases where certain of the argument are not needed, for instance in the *monotonicSequence* definition, it doesn't matter what comparison function is used if the sequence is empty or a singleton. Thus the definitions

*monotonicSequence* \_ [] = []

*monotonicSequence* \_ [x] = [[x]]

The other special character in argument definitions is `@`. It can be used to allocate a name to an entire complex construct, while allocating names to the entries as well. This is done in the output function *showSignFace*, as follows

*showSignFace* f@(s, \_, \_) = ...

whereby here *f* ends up referring to the entire *SignFace* it acts on, and *s* additionally refers to the sign of that face. The two `_` indicate that we do not wish to have names allocated to the partitions of the face.

Finally, we have the ability to construct type synonyms. By a declaration like

**type** *Newtype* = *Oldtype*

we can henceforth use *Newtype* as a type in its own right, which can be used anywhere where *Oldtype* could be used. We use this to make readable code, thus we can write, for instance

**type** *Sequence* = [*Int*]

**type** *Partition* = [*Sequence*]

**type** *Face* = (*Partition*, *Partition*)

*derivedFaces* :: *Face* → [*Face*]

instead of the more unwieldy und much less readable

*derivedFaces* :: ([[*Int*]], [[*Int*]]) → [[[[*Int*]], [[*Int*]]]]

**4.2. Lists and their manipulation.** A list type takes the form, on the type level, of [*Type*]. It consists of a randomly accessible sequence of entries, all of which has to have type *Type*. Lists can be given literally, by enclosing the comma-separated list of entries in square brackets, like [2, 3, 4]. An element can be prepended to a list using `:`, for example, 1:[2, 3, 4] = [1, 2, 3, 4]. For the reader with Lisp experience, `:` is very much like *cons*. The list [] is a special case – the empty list – and corresponds in Lisp terms to *Nil*. The integers between *a* and *b* can be listed by the construction [*a* .. *b*].

Note that strings are really lists of type [*Char*], so that "abc" = ['a', 'b', 'c']. This allows us to handle strings with the kind of toolbox we build for list manipulation as well.

When defining functions, we make large use of pattern matching on the input parameters. Thus, by stating several special cases and then a general case, we can achieve very compact and readable definitions. For this, the most important constructions are

- `[]`: The empty list.
- `_`: Any input parameter at all – we simply do not care for the value of this parameter, and will not use it in the definition.
- `[a]`: A singleton list, containing the element *a*.
- `(a : as)`: The element *a* followed by the rest of the list *as*. This can be expanded, so that `(a : b : c : etc)` picks out the first three elements of a list with at least three elements, and stores them in the variables *a*, *b* and *c* for the duration of the function definition.

Apart from the pattern matching, there is a wealth of functions available for handling lists. The ones used in the code here are

- `map`: applies a given function to all elements of a given list.
- `foldl'`: takes a binary function *\**, a first argument *a* and a list of arguments *a<sub>n</sub>*, and returns  $(\dots(((a * a_1) * a_2) * a_3) \dots) * a_n$ .
- `reverse`: reverses a finite list.
- `++`: concatenates two lists.
- `intersperse`: takes an element and inserts it between all elements of a list.
- `concat`: flattens a list of lists one level.
- `concatMap`: first applies a function to all elements of a list, and then flattens the result.
- `unlines`: intersperses the newline character, and concatenates the result to give a string representing each string in the list it is applied to on a new line on its own.
- `head`: returns the first element of a list.
- `length`: returns the number of elements of a list.
- `intersect`: returns the list of elements that occur in both the argument lists.
- `list !! i`: returns the element of index *i*. Thus `head list`  $\equiv$  `list !! 0`.
- `minimum`: returns the minimal element of a list of comparable elements.
- `maximum`: returns the maximal element of a list of comparable elements.
- `findIndices`: returns a list of all *i* such that the function given evaluates to *True* for the element of index *i*, and *False* for all other elements of the list.
- `findIndex`: returns *Just i* if *i* is the first element of the result from `findIndices`, and *Nothing* if `findIndices` would return an empty list.
- `null`: returns *True* if the argument is `[]` and *False* otherwise.
- `list \setminus list'`: removes all elements in `list'` from the list `list`.
- `take j list`: returns the list of the first *j* elements of `list`.
- `drop j list`: returns the list of the elements following the first *j* elements of `list`.
- `filter`: returns only those elements for which the function given evaluates to *True*.
- `tails list`: is equivalent to `map (\j → drop j list) [1..length list]`.
- `nub`: removes duplicates from a list.
- `sort`: sorts the list.
- `elem`: checks whether an element appears in a list.
- `sum`: sums the elements of a list.

**4.3. Maybe – graceful error handling.** It is quite possible that a calculation can, or may, be expected to fail on some input. The canonical way to handle this in Haskell is to assign a type to the results of the calculation that handles error

cases. Unless an error description is necessary, the *Maybe* type encapsulation is what gets used. The *Maybe Mytype* has two different kinds of elements – either *Nothing* or *Just aValue*. The return value of *Nothing*, thus, is used to signify a failed calculation.

In order to handle output from this kind of calculation, there are a few methods available.

**Pattern matching:** with the patterns *Nothing* or *Just a* can be used to separate into cases depending on the results of the calculation.

*isNothing:* returns *True* if the object considered is *Nothing*.

*isJust:* returns *True* if the object considered is not *Nothing*.

*fromJust:* takes *Just a* and returns *a*. If used on *Nothing* causes an exception to be thrown, thus aborting execution entirely. To be used only when it is impossible to receive a *Nothing*.

*mapMaybe:* maps a function returning *Maybe Sometype* over a list, and returns a list of only the results that were encapsulated in *Just*.

**4.4. Pairs of elements.** The expression  $(a, b)$  gives an element of type  $(A\text{Type}, B\text{Type})$ .

This is a way to construct static length tuples of elements that need not be of the same type – as different from the lists, where all elements must be of the same type. The functions *fst* and *snd* allow access to *a* and *b* respectively. Pattern matching works in the expected way –  $(a, b)$  assigns the name *a* to the first element and *b* to the second in a pattern match.

**4.5. Data.Map – associative arrays.** An associative array is a way to store data with other indexing sets than just the non-negative integers. Thus, we can use (almost) any indexing type in an associative array. The application we use it for here is for the type carrying linear combinations of faces. Here, a linear combination is an allocation from the set of occurring faces to the coefficients for each face.

In order to interact with these constructs, there are a number of operations available – many of which are not used in the code, but will prove useful to any user who wants to extract information from a calculation.

*signFaceList:* converts back from the linear combinations to a list of signed faces.

*showLinearCombination:* produces a string representing the entire linear combination.

*Map.size:* gives the number of terms in a linear combination, though not necessarily excluding terms with coefficient 0.

*Map.filter:* removes terms for which the value held evaluates to *False* in a given function. Thus *Map.filter* ( $\neq 0$ ) can be used to remove all trivial terms from the results.

$(\text{permutahedronDiagonal } n) ! \text{someFace}:$  returns the sign of *someFace*.

*Map.elems:* returns all the coefficients in the diagonal.

*Map.keys:* returns all the faces in the diagonal.

**4.6. Logical and arithmetic operations.** Logic has much of the expected operators, thus we have

$\wedge$ : for logical and,

$\vee$ : for logical or,

$\neg$ : for logical negation,

$\equiv$ : for equality testing,  
 $\neq$ : for inequality testing,  
 $\geq, >, \leq$  **and**  $<$ : for orderings,  
 $+$ : for addition,  
 $*$ : for multiplication,  
 $\uparrow$ : for exponentiation and  
 $-$ : for subtraction or negation, depending on context.

**4.7. Loose ends.** Finally, we mention several purely syntactical subtleties. The keyword **where** can be used to postpone definition of symbols used until an expression is completely given. Thus, anything listed after the **where** becomes definitions that scope over the expression **where** follows.

The **if-then-else** constructs give little surprise, but are a language construct of their own, specializing expressions on the form

**case something of**

*value1*  $\rightarrow$  *result1*

*value2*  $\rightarrow$  *result2*

where the choice of result is made over a larger array of alternatives than only *True* and *False*.

The special operators  $\circ$  and  $\$$  handle function composition. The operation  $\circ$  works entirely as expected, with  $f (g (h x)) = (f \circ g \circ h) x$ .  $\$$  on the other hand works more as function application, and has a kind of backwards associativity, so that instead of writing  $f (someLargeExpression)$  we can content ourselves by writing  $f \$ someLargeExpression$ .

Since, in functional programming, functions are just as good arguments to other functions as any other sort of value, there is a need for some way to construct an anonymous, lightweight, throwaway function to be inserted in some particular context or other. This is for historical reasons done with the  $\lambda$  construction. By an expression of the form  $\lambda x \rightarrow someExpression x$  we declare a function that takes an argument  $x$  and returns *someExpression*  $x$ .

## 5. ACKNOWLEDGEMENTS

This paper would not have been possible without all the fruitful conversations held with Ron Umble. The author is thankful for these and the generous help given in preparing this preprint.

The author further acknowledges that the ideas would have remained largely unpublished without the urging by Jim Stasheff to write it up.

The Haskell section has enjoyed the benefits of several experienced Haskell programmers, most notably Shae Erisson.

The author would like to acknowledge his supervisor David J. Green, who has remained supportive of the author's ideas and efforts.

## REFERENCES

- [GHC99] GHC – The Glasgow Haskell Compiler. <http://haskell.org/ghc>, January 1999.
- [JHA<sup>+</sup>99] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999. Available at <http://www.haskell.org>.

- [SU04] Samson Saneblidze and Ronald Umble. Diagonals on the permutahedra, multiplihedra and associahedra. *Homology Homotopy Appl.*, 6(1):363–411 (electronic), 2004.
- [Ton97] Andy Tonks. Relating the associahedron and the permutohedron. In *Operads: Proceedings of Renaissance Conferences (Hartford, ct/Luminy, 1995)*, volume 202 of *Contemp. Math.*, pages 33–36, Providence, RI, 1997. Amer. Math. Soc.
- [Ton07] Andy Tonks. Personal communication, 2007.
- [Wea05] Steve Weaver. Computing the Saneblidze-Umbler diagonal on permutahedra. Senior thesis, Millersville University, May 2005.

*E-mail address:* `mik@math.uni-jena.de`

LEHRSTUHL FÜR ALGEBRA UND ZAHLENTHEORIE, MATHEMATISCHES INSTITUT, FAKULTÄT FÜR  
MATHEMATIK UND INFORMATIK, FSU JENA, 07737 JENA, GERMANY

---

**Example 1** Some calculations with the `SaneblidzeUmble.lhs` implementation
 

---

```
$ ghci SaneblidzeUmbleSigns.lhs

--- ---
/ _ \ /\ /\ _ _(_)
/ /_\\ /_ / / | | GHC Interactive, version 6.6, for Haskell 98.
/ /_\\ _ / /___| | http://www.haskell.org/ghc/
\___/\ /_/\___/|_| Type :? for help.

Loading package base ... linking ... done.
[1 of 1] Compiling SaneblidzeUmbleSigns ( SaneblidzeUmbleSigns.lhs, interpreted )
Ok, modules loaded: SaneblidzeUmbleSigns.
*SaneblidzeUmbleSigns> Map.size . permutahedronDiagonal $ 4
50
*SaneblidzeUmbleSigns> Map.size . associahedronDiagonal $ 4
22
*SaneblidzeUmbleSigns> map (Map.size . permutahedronDiagonal) [1..6]
[1,2,8,50,432,4802]
*SaneblidzeUmbleSigns> map (Map.size . associahedronDiagonal) [1..6]
[1,2,6,22,91,408]
*SaneblidzeUmbleSigns> putStr . unlines . map showSignFace .
signFaceList . permutahedronDiagonal $ 3
+1|2|3x1,2,3
-1|2,3x3|1,2
-1|2,3x1,3|2
+1,2|3x2,3|1
+1,2|3x2|1,3
+1,2,3x3|2|1
-1,3|2x3|1,2
+2|1,3x2,3|1
*SaneblidzeUmbleSigns> putStr . unlines . map showSignFace .
signFaceList . associahedronDiagonal $ 3
+1|2|3x1,2,3
-1|2,3x3|1,2
+1,2|3x2,3|1
+1,2|3x2|1,3
+1,2,3x3|2|1
+2|1,3x2,3|1
*SaneblidzeUmbleSigns> :q
Leaving GHCi.
```

---